Skiplist.

class Node

public:

① 构造 { Node() {};
 Node(K, V, level)

② 析构

③ { get_key()
 get_value()
 set_value()

④ Node<K, V> **forward

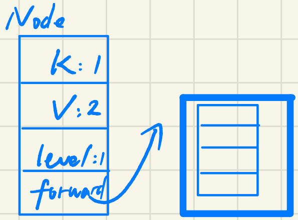
二级指针.可分析为指向一个数组?

⑤ node_level.

private:

K key;

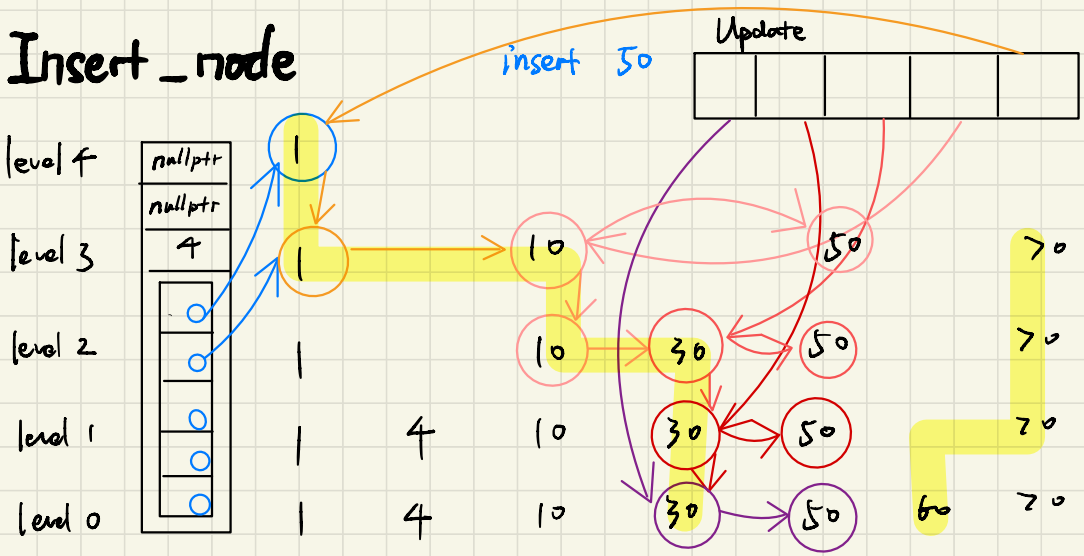
V value;



int *[5] 便是指针数组

this → forward = new Node<k, V> * [level+1];

Insert_node



this → _header = new Node<K, V>(K, V, _max_level)

以上是初始化工作

Node<K, V> *update[_max_level + 1];

用于定制的数组？

---

for (

)

while (current → forward[i] != NULL && current → forward[i] → get_key() < key)

current = current → forward[i];

update[i] = current;

使用 update 数组，记录待插入到的边界项，如图

情况1.

get_key()

== key

已经在

直接取锁完

成插入任务.

情况2

int random_level = get_random_level();

if (random_level > _skip_list_level) ...

创建节点，并进行节点的前后连接

```
int k = 1;
while (rand() % 2)
    k++;
k = (k < max_level)
 ? k : _max_level;
return k;
```

Stress_test

创建多线程

```
for (i=0; i< NUM_THREADS; i++)
    rc = pthread_create (&threads[i], NULL, insertElement, (void*)i);
...
```

多线程运行

```
for (
    if ( pthread_join ( threads[i], &ret) != 0 )
    ...
```

insertElement

Skiplist 性能分析

摘要描述

跳表在动态查找过程中使用了一种非严格的平衡机制来让插入和删除都更加便利和快捷。这种非严格平衡是基于概率的，而不是平衡树的严格平衡。

复杂度分析

① 确定最高索引层数 m

第 m 级索引 $\frac{n}{2^m}$　　$m = \log_2 n$

遍历的复杂度 $O(d * \log n)$　　d 是每一层要遍历的结点数

② 空间存储

$2 + 4 + 6 + \cdots + \frac{n}{8} + \frac{n}{4} + \frac{n}{2} = n - 2$

额外空间 $O(n-2)$

Redis 实现应用

```
int zslRandomLevel(void)
    int level = 1;
    while ((random() & 0xFFFF) < (ZSKIPLIST_P * 0xFFFF))    // 0.25
        level += 1;
    return (level < ZSKIPLIST_MAXLEVEL) ? level : Z...L)     // 32
```

true 的概率为 1/4.

节点平均层数

$$L_{avg} = 1 \times (1-p) + 2p(1-p) + 3p^2(1-p) + \cdots + kp^{k-1}(1-p)$$

$$= (1-p)(1 + 2p + 3p^2 + \cdots + kp^{k-1})$$

$$= \frac{1}{1-p}.$$